INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Plug-in Scheduler Design for a Distributed Grid Environment

Eddy Caron — Andréea Chis — Frédéric Desprez — Alan Su

N° 6030

Novembre 2006

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--6030--FR+ENG

Plug-in Scheduler Design for a Distributed Grid Environment

Eddy Caron , Andréea Chis , Frédéric Desprez , Alan Su

Thème NUM — Systèmes numériques
Projet GRAAL

Rapport de recherche n° 6030 — Novembre 2006 — 15 pages

Abstract: This report presents the approach chosen within the DIET (Distributed Interactive Engineering Toolbox) project a Grid-RPC environment to allow a resource broker to be tuned for specific application classes. Our design allows the use of generic or application dependent performance measures in a simple and seamless way.

Key-words: Grid Computing, Scheduling, Performance Prediction

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Plug-in Scheduler Design for a Distributed Grid Environment

Résumé : Ce rapport présente l'approche choisie dans le cadre du projet DIET (Distributed Interactive Engineering Toolbox), un environnement de type Grid-RPC, permettant à un ordonnanceur d'être adapté pour des classes d'applications spécifiques. Notre approche permet, de façon simple, d'utiliser des mesures de performances génériques ou dépendantes de l'application.

Mots-clés : Calcul sur Grille, Ordonnancement, Prédiction de performance

1 Introduction

The GridRPC approach [14] where remote servers execute computation requests on behalf of clients connected through the Internet is one of the most flexible and efficient solutions for porting large scale applications over the grid. In this model resource brokers (also called agents) are responsible for identifying the most appropriate servers for a given set of requests.

One of the main problems is, of course, to evaluate the costs associated with remote execution of these requests. Depending on the target application, several measurements have to be taken into account. Execution time is typically the measure with which users are most concerned, but others factors, such as available memory, disk, machine load, batch queue length may also be significant. Moreover, particular applications may have specific features and behaviors, which should also be considered. However, these various performance measures can only be taken into account if the middleware allows tuning of its internal scheduling software by the applications it services.

This paper presents the approach chosen within the DIET (Distributed Interactive Engineering Toolbox) project to allow a resource broker to be tuned for specific application classes. Our design allows the use of generic or application dependent performance measures in a simple and seamless way. The remainder of the paper organized as follows: Section 3 presents the architecture of the DIET middleware framework. Section 4 describes the plug-in scheduler feature and Section 5 describes the CoRI collector which allows the management of different performance measures. Finally, Section 6 presents some early experiments of this new feature of our GridRPC framework before a conclusion.

2 Related work

Several middleware frameworks follow the GridRPC API from the GGF like Ninf [10] or GridSolve [19], but none of them expose interfaces that allow their scheduling internals to be tuned for specific application classes.

APST [4] allows some modifications of the internals of the scheduling phase, mainly to be able to choose the scheduling heuristic. Several heuristics can be used like Max-min, Min-min, or X-sufferage.

Scheduling heuristics for different application classes have been designed within the AppLeS [5] and GrADS [1] projects. GrADS is built upon three components [6]. A *Program Preparation System* handles application development, composition, and compilation, a *Program Execution System* provides on-line resource discovery, binding, application performance monitoring, and rescheduling. Finally, a *binder* performs a resource-specific compilation of the intermediate representation code before it is launched on the available resources. One interesting feature of GrADS is that it provides application specific performance models. Depending of the target application, these models can be built upon an analytic evaluation by experts joined with empirical models obtained by real world experiments. For some applications, simulation is used instead to guess their behavior on some specific architectures.

These models are then fed into the schedulers to find the most appropriate resources used to run the application.

Recently, some work has been done to be able to cope with dynamic platform performance at the execution time [13, 17]. These approaches allows an algorithm to automatically adapt itself at run-time depending of the performance of the target architecture, even if it is heterogeneous.

Within the Condor project [16], the ClassAds language was developed [12]. This language allows to specify resource query requests with attributes built upon lists of constants, arbitrary collections of constants and variables combined with arithmetic and logic operators. The result is a multicriteria decision problem. The approach presented in our paper allows a scheduling framework to offer useful informations to a metascheduler like Condor.

3 DIET Aim and Design Choices

The DIET component architecture is structured hierarchically for improved scalability. Such an architecture is flexible and can be adapted to diverse environments, including arbitrary heterogeneous computing platforms. The DIET toolkit [3] is implemented in CORBA and thus benefits from the many standardized, stable services provided by freely-available and high performance CORBA implementations. CORBA systems provide a remote method invocation facility with a high level of transparency. This transparency should not dramatically affect the performance, communication layers being well optimized in most CORBA implementations [7]. These factors motivate our decision to use CORBA as the communication and remote invocation fabric in DIET.

Our framework comprises several components. A **Client** is an application that uses the DIET infrastructure to solve problems using an RPC approach. Clients access DIET via various interfaces: web portals, PSEs such as SCILAB, or programmatically using published C or C++ APIs. A **SeD**, or server daemon, acts as the service provider, exporting functionality via a standardized computational service interface; a single SeD can offer any number of computational services. A SeD can also serve as the interface and execution mechanism for either a stand-alone interactive machine or a parallel supercomputer, by interfacing with its batch scheduling facility. The third component of the DIET architecture, **agents**, facilitate the service location and invocation interactions of clients and SeDs. Collectively, a hierarchy of agents provides higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a single **Master Agent (MA)** and several **Local Agents (LA)**. Figure 1 shows an example of a DIET hierarchy.

The **Master Agent** of a DIET hierarchy serves as the distinguished entry point from which the services contained within the hierarchy may be logically accessed. Clients identify the DIET hierarchy using a standard CORBA naming service. Clients submit requests – composed of the name of the specific computational service they require and the necessary arguments for that service – to the MA. The MA then forwards the request to its children, who subsequently forward the request to their children, such that the request is

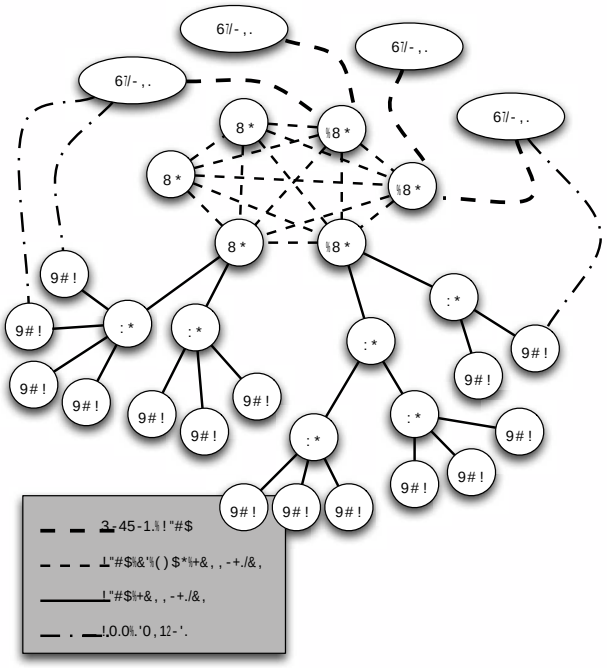

Figure 1: DIET hierarchical organization.

eventually received by all SeDs in the hierarchy. SeDs then evaluate their own capacity to perform the requested service; capacity can be measured in a variety of ways including an application-specific performance prediction, general server load, or local availability of datasets specifically needed by the application. SeDs forward their responses back up the agent hierarchy. Based on the capacities of individual SeDs to service the request at hand, the agents collectively reduce the set of server responses to a manageable list of potential server choices. This list is sorted using an objective function (e.g., computation cost, communication cost, or machine load) that is appropriate for the application. The client program may then submit the request directly to any of the proposed servers, though typically the first server will be preferred as it is predicted to be the most appropriate server. The scheduling strategies used in DIET are the subject of this paper.

Finally, NES environments like Ninf [10] and GridSolve [19] use a classic socket communication layer. Nevertheless, several problems to this approach have been pointed out such as the lack of portability or the limitation of opened sockets. A distributed object environment, such as CORBA has been shown to be a good base for building applications that manage access to distributed services in heterogeneous environments[8]. First and foremost, CORBA

provides platform-independent communication primitives, as well as useful infrastructure components for large-scale deployments of distributed applications. Moreover, CORBA systems provide a remote method invocation facility with a high level of transparency. This transparency should not dramatically affect the performance, as the communication layers in most CORBA implementations have been highly optimized [7].

4 Plug-in Scheduler

A first version of scheduling in DIET was based on the FIFO principle - a task submitted by a client was scheduled on the SeD whose response to the service query arrived the first to the MA. This approach was not taking into consideration any local condition that held at the SeD level. A second version allowed a mono-criteria scheduling based on application-specific performance predictions. Later a round-robin scheduling scheme was implemented in DIET, resulting in good performance for task distribution over a homogeneous platform, which wasn't however suitable in the case of a heterogeneous one.

Applications targeted for the DIET platform are now able to exert a degree of control over the scheduling subsystem via *plug-in schedulers*. As the applications that are to be deployed on the grid vary greatly in terms of performance demands, the DIET plug-in scheduler facility permits the application designer to express application needs and features in order that they be taken into account when application tasks are scheduled. These features are invoked at runtime after a user has submitted a service request to the MA, which broadcasts the request to its agent hierarchy.

When an application service request arrives at a SeD, it creates a *performance estimation vector* – a collection of *performance estimation values* that are pertinent to the scheduling process for that application. The values to be stored in this structure can be either values provided by CoRI (Collectors of Resource Information) described in Section 5, or custom values generated by the SeD itself. The design of the estimation vector subsystem is modular; future performance measurement systems can be integrated with the DIET platform in a fairly straightforward manner.

CoRI generates a basic set of performance estimation values, which are stored in the estimation vector and identified by system-defined tags; Table 1 lists the tags that may be generated by a standard CoRI installation. Application developers may also define performance values to be included in a SeD response to a client request. For example, a DIET SeD that provides a service to query particular databases may need to include information about which databases are currently resident in its disk cache, in order that an appropriate server be identified for each client request. By default, when a user request arrives at a DIET SeD, an estimation vector is created via a default estimation function; typically, this function populates the vector with standard CoRI values. If the application developer includes a custom *performance estimation function* in the implementation of the SeD, the DIET framework will associate the estimation function with the registered service. Each time a user request is received by a SeD associated with such an estimation function, that

function, instead of the default estimation procedure, is called to generate the performance estimation values.

Information tag starts with EST_	multi-value	Explanation
TCOMP		the predicted time to solve a problem
TIMESINCELASTSOLVE		time since last solve has been made (sec)
FREECPU		amount of free CPU between 0 and 1
LOADAVG		CPU load average
FREEMEM		amount of free memory (Mb)
NBCPU		number of available processors
CPUSPEED	x	frequency of CPUs (MHz)
TOTALMEM		total memory size (Mb)
BOGOMIPS	x	the BogoMips
CACHECPU	x	cache size CPUs (Kb)
TOTALSIZEDISK		size of the partition (Mb)
FREESIZEDISK		amount of free place on partition (Mb)
DISKACCESREAD		average time to read from disk (Mb/sec)
DISKACCESWRITE		average time to write to disk (Mb/sec)
ALLINFOS	x	[empty] fill all possible fields

Table 1: Explanation of the estimation tags

In the performance estimation routine, the SeD developer should store in the provided estimation vector any performance data needed by the agents to evaluate server responses. Such vectors are then the basis on which the suitability of different SeDs for a particular application service request is evaluated. Specifically, a local agent gathers responses generated by the SeDs that are its descendents, sorts those responses based on application-specific comparison metrics, and transmits the sorted list to its parent. The mechanics of this sorting process comprises an *aggregation method*, which is simply the logical process by which SeD responses are sorted. If application-specific data are supplied (i.e., a custom estimation function has been specified), an alternative method for aggregation is needed. Currently, a basic priority scheduler has been implemented, enabling an application developer to specify a series of performance values that are to be optimized in succession. From the point of view of an agent, the aggregation phase is essentially a sorting of the server responses from its children. A priority scheduler logically uses a series of user-specified tags to perform the pairwise server comparisons needed to construct the sorted list of server responses.

5 CoRI

As we have seen in the previous section, the scheduler requires performance measurement tools to make effective scheduling decisions. Thus, DIET depends on reliable grid resource information services. In this section we introduce the exact requirements of DIET for a grid information service, the architecture of the new tool CoRI (Collectors of Resource Information) and the different components inside of CoRI.

5.1 CoRI architecture

In this section, we describe the design of CoRI, this new platform performance subsystem that we have implemented to enable future versions of the DIET framework to more easily interface with third-party performance monitoring and prediction tools. Our goal is to facilitate the rapid incorporation of such facilities as they emerge and become widely available. This issue is especially pertinent, considering the fact that DIET is designed to run on heterogeneous platforms, on which many promising but immature tools may not be universally available. Such a scenario is common, considering that many such efforts are essentially research prototypes. To account for such cases, we have designed the performance evaluation subsystem in DIET to be able to function even in the face of varying levels of information in the system.

We designed CoRI to ensure that it (i) provides timely performance information in order to avoid impeding the scheduling process and (ii) presents a general-purpose interface capable of encapsulating a wide range of resource information services. Firstly, it must provide basic measurements that are available regardless of the state of the system. The service developer can rely on such information even if no other resource performance prediction service like FAST [11] (Fast Agent's System Timer) or NWS [18] is installed. Secondly, the tool must **manage** the simultaneous use of different performance prediction systems within a single heterogeneous platform. To address these two fundamental challenges, we offer two solutions: the **CoRI-Easy** collector to universally provide basic performance information, and the **CoRI Manager** to mediate the interactions among different collectors. In general, we refer collectively to both these solutions as the **CoRI** tool, which stands for Collectors of Resource Information. Both subsystems are described in the following section.

Broadly, the CoRI-Easy facility is a set of simple requests for basic resource information, and the CoRI Manager is the tool that enables application developers to add access methods to other resource information services. As CoRI-Easy is fundamentally just a resource information service, we implement it as a collector that is managed by the new CoRI Manager. Note that CoRI-Easy is not the only collector available; for example, FAST can be used as well. Moreover, adding new collectors is a simple matter of implementing a thin interface layer that adapts the performance prediction data output to the reporting API that the CoRI Manager expects.

5.2 CoRI Manager

The CoRI Manager provides access to different **collectors**, which are software components that can provide information about the system. This modular design decouples the choice of measurement facilities and the utilization of such information in the scheduling process. Even if the manager should aggregate across performance data originating from different resource information services, the raw trace of data remains, and so its origin can be determined. For example, it could be important to distinguish the data coming from the CoRI-Easy collector and the FAST collector, because the performance prediction approach that FAST uses is more highly tuned to the features of the targeted application. Furthermore, the modular design

of the CoRI Manager also permits a great deal of extensibility, in that additional collectors based on systems such as Ganglia [9] or NWS [18] can be rapidly implemented via relatively thin software interface layers. This capability enables DIET users to more easily evaluate prototype measurement systems even before they reach full production quality.

5.3 Technical overview of CoRI

In this section, we describe in greater detail the structure of the *estimation vector*. We then enumerate the standard metrics of performance used in DIET and present the various CoRI Manager functions. The vector is divided into two parts. The first part represents "native" performance measures that are available through CoRI (e.g., the number of CPUs, the memory usage, etc.) and the scheduling subsystem (e.g., the time elapsed since a server's last service invocation). The second part is reserved for developer-defined measurements that are meaningful solely in the context of the application being developed. The vector supports the storage of single and multiple values, because some performance prediction measurements are not only single values (called scalars) but a list of values (e.g., the load on each processor of a multi-processor node). An estimation vector is essentially a container for a complete performance snapshot of a server node in the DIET system, composed of multiple scalars and lists of performance data.

Figure 2 is an estimation vector example that uses different measurement types that are identified by tags, which are described in greater detail in the next section. In this example vector in particular, data representing the number of CPUs,

tag	scalar	0	1
EST_NBCPU	2		
EST_CPUSPEED		1400	2000
0	25		
1		1	0

Figure 2: An example of the estimation vector with some estimation tags.

To differentiate among the various classes of information that may be stored in an estimation vector, a *data tag* is associated with each scalar value or list of related values. This tag enables DIET's performance evaluation subsystems to identify and extract performance data. There are two types of tags: system tags and user-defined tags. System tags correspond to application-independent data that are stored and managed by the CoRI Manager; Table 1 enumerates the set of tags that are supported in the current DIET architecture. User-defined tags represent application-specific data that are generated by specialized performance estimation routines that are defined at compile-time for the SeD. Note also that

the `EST_ALLINFOS` tag is in fact a pseudo-tag that is simply used to express a request for all performance information that is available from a particular collector. At the moment of service registration, a SeD also declares a priority list of comparison operations based on these data that logically expresses the desired performance optimization semantics. The mechanisms for declaring such an optimization routine is outside the scope of this paper; for a fully detailed description of the API, please consult the DIET website [15].

The basic public interface of the CoRI Manager that is available to DIET service developers (and implicitly to DIET developer as well) consists of three functions. The first function allows the initialization of a given collector and adds the collector to the set of collectors that are under the control of the CoRI Manager. The second function provide access to measurements. The last function tests the availability of the CoRI-Easy collector.

5.4 CoRI-Easy collector

The CoRI-Easy collector is a resource collector that provides basic performance measurements of the SeD. Via the interface with the CoRI Manager, the service developer and the DIET developer are able to access CoRI-Easy metrics. We first introduce the basic design of CoRI-Easy, and then we will discuss some specific problems. CoRI-Easy should be extensible like CoRI Manager, i.e. the measurement functions must be easily replaceable or extended by new functionality as needed.

Consequently, we use a functional approach: functions are categorized by the information they provide. Each logical type of performance information is identified by an *information class*, which enables users to simply test for the existence of a function providing a desired class of information. Thus, it is even possible to query the CoRI-Easy collector for information about the platform that may not yet have been realized. Our goal was not to create another sensor system or monitor service; CoRI-Easy is a set of basic system calls for providing basic performance metrics. Note that the data measured by CoRI-Easy are available via the interface of the CoRI Manger.

CPU evaluation CoRI-Easy provides CPU information about the node that it monitors: the *number of CPUs*, the *CPU frequency* and the *CPU cache size*. These static measurements do not provide a reliable indication of the actual load of CPUs, so we also measure the node's *BogoMips* (a normalized indicator of the CPU power), the *load average* and *CPU utilization* for indicating the CPU load.

Memory capacity The memory is the second important factor that influences performance. CoRI monitors the *total memory size* and the *available memory size*.

Disk Performance and capacity CoRI-Easy also measures the *read and write performance* of any storage device available to the node, as well the *maximal capacity* and the *free capacity* of any such device.

Network performance CoRI-Easy should monitor the performance of interconnection networks that are used to reach other nodes, especially those in the DIET hierarchy to which that node belongs; this functionality is planned for a future release.

6 Experimentation

The first experiment we conducted compares a basic application-independent round robin task distribution with a distribution obtained by a scheduler that accounts for processor capacities (both static and dynamic) for computationally intensive homogeneous tasks. A platform comprising 1 MA, 2 LAs and 6 SeDs (each LA having 3 associated SeDs) was deployed on a heterogeneous platform. The computationally intensive task consisted of several runs of the DGEMM (i.e. matrix-matrix multiplication) function from BLAS (Basic Linear Algebra Subprograms) library.

The RR scheduler (Round Robin scheduler) is a priority scheduler that aggregates SeD responses based on the time elapsed since the last execution start (the `EST_TIMESINCELASTSOLVE` CoRI tag), whereas the CPU scheduler is a priority scheduler that maximizes the ratio $\frac{BOGOMIPS}{1+load\_average}$ (the `EST_BOGOMIPS` and `EST_LOADAVG` tags respectively). The BOGOMIPS metric characterizes the raw processor speed, and the load average provides an indication of CPU contention among runnable processes – in this case an estimation over the last minute. The different CPU speeds of the resources were simulated by running the DGEMM function several times and adapting the BOGOMIPS value obtained through CoRI accordingly.

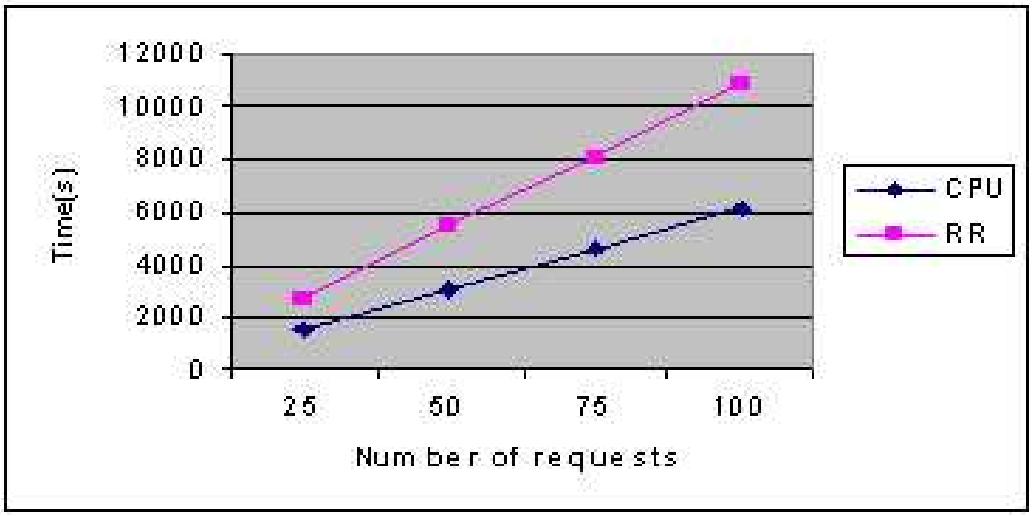

Figure 3: CPU vs RR scheduler - total computation time.

The behavior of the two schedulers was studied for requests spaced at different time intervals – here we focus on requests spaced by 1 minute. The results show that the CPU-based scheduling policy provides an equitable allocation of platform resources, as overall,

the run times for the tasks are almost equal. Conversely, the round robin scheduler was unable to balance the aggregate load of the tasks: some completed quickly whereas those assigned to heavily loaded processors were delayed substantially. As seen in Figure 3, the total runtime of the system under varying loads is consistently better when using the CPU scheduler, relative to the performance observed with the RR scheduler.

We conducted a second experiment that illustrates the difference between the task distribution on a round robin basis versus a distribution based on the criterion of disk access speed. We hypothesize that the former will result into degraded performance due to disk device contention for I/O-intensive tasks.

We used the same platform as we described in our previous experiment, (1 MA, 2 LAs and 6 SeDs) while the load is composed of tasks that executed multiple disk writes (according to the disk write speed to be simulated) and the same number of DGEMM calls. In order to truly test the hypothesis, we designed the tasks such that the disk access time be significantly greater than the DGEMM computing time.

In order to model a heterogeneous platform, the disk speed estimations retrieved by CoRI were scaled by a factor and the number of disk writes performed by the associated service function was scaled accordingly.

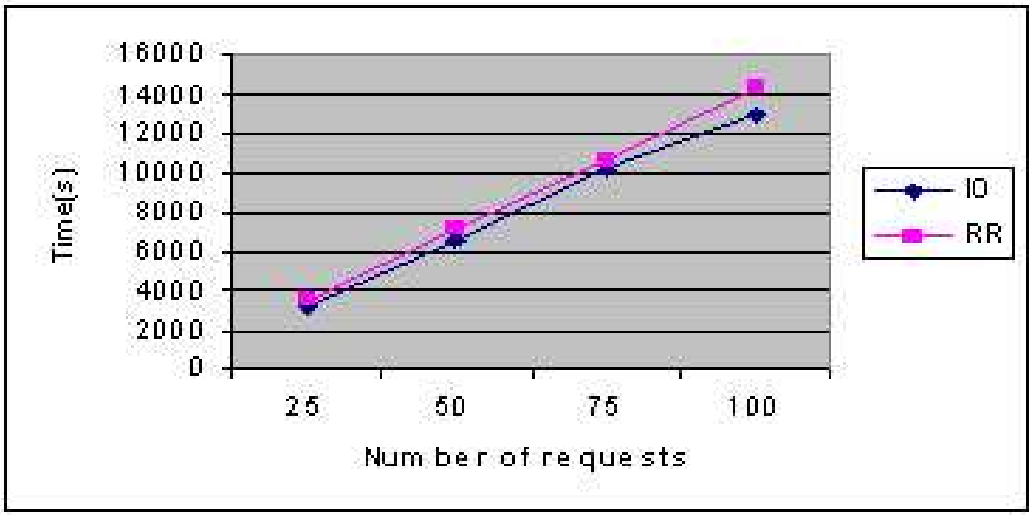

Figure 4: I/O vs RR scheduler - total computation time.

The RR scheduler is identical to the one in the first experiment - a priority scheduler that maximizes the time elapsed since the last execution start (`EST_TIMESCINCELASTSOLVE` CoRI tag) whereas the I/O scheduler is a priority scheduler that maximizes the disk write speed (`EST_DISKACCESWRITE` CoRI tag). The behavior of the two schedulers was studied for

requests spaced at different time intervals – here we focus on requests spaced at 35 seconds. As seen in Figure 4, the runtime on the platform is improved by using the I/O scheduler.

Though these performance results are not unexpected, the ease with which these experiments were conducted is an indication of the utility of the resource performance gathering and plug-in scheduling facilities we have implemented. Prior to their introduction, similar results were attainable on comparable (and indeed more complex) applications through the FAST performance prediction system. However, the framework described in this paper holds a distinct advantage over the capabilities previously available to DIET application developers: the "barriers to entry" to realizing practical application-tuned scheduling have been dramatically lowered. Previously, FAST required for maximal efficacy that the application's performance be benchmarked for a range of parameterizations, even if the application execution behavior was well-understood. The plug-in facilities enable this information to be readily encoded with the application, eliminating the need for potentially expensive microbenchmarks. Moreover, the CoRI resource performance collection infrastructure enables middleware maintainers to closely assess and control the intrusiveness of the monitoring subsystem. By configuring CoRI to gather only that information that is needed for the applications that a specific DIET hierarchy is meant to support, excessive monitoring costs are effectively avoided. Collectively, these advantages represent a substantial improvement in DIET's application monitoring and scheduling capabilities.

The performance estimations provided for scheduling are suitable for dedicated resource platforms with batch scheduler facilities such as Grid'5000 [2]. On a shared resource environment, there could appear differences between the estimations obtained at the SeD selection moment and the values existing when the client-SeD communication begins. However, even though in some cases the information could be erroneous, an informed decision is preferable.

7 Conclusion

In this paper, we described the design of a plug-in scheduler in a grid environment, and reported on an implementation of this design in the DIET toolkit. To provide underlying resource performance data needed for plug-in schedulers, we designed a tool that facilitates the management of different performance measures and different kinds of resource collector tools. As a proof of concept, we developed a basic resource information collector that enabled very basic performance data to be incorporated into DIET's scheduling logic. We then conducted two experiments that illustrated the potential benefits of utilizing such information in realistic distributed computing scenarios. As future work, we plan to use the plug-in scheduler to design highly-tuned application-specific schedulers for several real applications that we have begun to study. We also intend to incorporate new collectors that seem promising in terms of providing resource performance data that may be useful in the DIET scheduling process.

Acknowledgment

We would like to give a very special thanks to Peter Frauenkron for his help on the design and implementation of CoRI. Diet was developed with financial support from the French Ministry of Research (RNTL GASP and ACI ASP) and the ANR (Agence Nationale de la Recherche) through the LEGO project referenced ANR-05-CIGC-11.

References

[1] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications*, 15(4):327–344, November 2001.

[2] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid '05)*, Seattle, Washington, USA, November 2005.

[3] E. Caron and F. Desprez. Diet: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[4] H. Casanova and F. Berman. *Grid Computing*, chapter Parameter Sweeps on the Grid with APST. Wiley Series in Communications Networking & Distributed Systems. Wiley, 2003.

[5] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The appleS parameter sweep template: User-level middleware for the grid. *Scientific Programming*, 8(3):111–126, 2000.

[6] H. Dail, O. Sievert, F. Berman, H. Casanova, S. YarKahn, J. Dongarra, C. Liu, L. Yang, D. Angulo, and I. Foster. *Grid Resource Management*, chapter Scheduling in the Grid Application Development Software Project, pages 73–98. Kluwer Academic Publisher, September 2003.

[7] A. Denis, C. Perez, and T. Priol. Towards high performance CORBA and MPI middlewares for grid computing. In Craig A. Lee, editor, *Proc. of the 2nd International Workshop on Grid Computing*, number 2242 in LNCS, pages 14–25, Denver, Colorado, USA, November 2001. Springer-Verlag.

[8] M. Henning and S. Vinoski. *Advanced CORBA(R) Programming with C++*. Addison-Wesley Pub Co, 1999.

[9] M. Massie, B. Chun, and D. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004.

[10] H. Nakada, M. Sato, and S. Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *FGCS, Metacomputing Issue*, 15(5-6):649–658, 1999. http://ninf.apgrid.org/papers/papers.shtml.

[11] M. Quinson. Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment. In *International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02), in conjunction with IPDPS'02*, Apr 2002.

[12] R. Raman, M. Solomon, M. Livny, and A. Roy. *The ClassAds Language*, chapter Scheduling in the Grid Application Development Software Project, pages 255–270. Kluwer Academic Publisher, September 2003.

[13] D.A. Reed and C.L. Mendes. Intelligent Monitoring for Adaptation in Grid Applications. In *Proceedings of the IEEE*, volume 93, pages 426–435, February 2005.

[14] K. Seymour, C. Lee, F. Desprez, H. Nakada, and Y. Tanaka. The End-User and Middleware APIs for GridRPC. In *Workshop on Grid Application Programming Interfaces, In conjunction with GGF12*, Brussels, Belgium, September 2004.

[15] GRAAL Team. DIET User's Manual. http://graal.ens-lyon.fr/DIET.

[16] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[17] S. Vadhiyar and J. Dongarra. Self Adaptability in Grid Computing. *Concurrency and Computation: Practice and Experience*, 17(2-4), 2005.

[18] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems, Metacomputing Issue*, 15(5-6):757–768, Oct. 1999.

[19] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, and J. Dongarra. Recent developments in gridsolve. *International Journal of High Performance Computing Applications*, 20(1):131–141, 2006.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399